Functional and Non Functional Requirements:

Let's break down the functional and non-functional requirements using a real-world system example: an Online Food Delivery Platform (like Swiggy or Zomato).

✅ Functional Requirements (FRs)

Functional requirements define **what the system should do** — the features, capabilities, and interactions.

🔹 Examples:

Requirement	Description
User Registration/Login	Users must be able to register and log in securely.
Search Restaurants	Users can search for restaurants based on location, cuisine, or name.
Browse Menu	Users can view restaurant menus, prices, and item availability.
Place Order	Users can select items, add to cart, and place an order.
Track Order	Users can view the current status of their orders (e.g., preparing, out for delivery).
Payment Processing	Users can pay using UPI, card, or wallet services.
Delivery Assignment	The system should assign a delivery partner based on location and availability.

✅ Non-Functional Requirements (NFRs)

Non-functional requirements define **how the system should behave** — performance, scalability, usability, etc.

🔹 Examples:

Requirement	Description
Scalability	The system should handle 1 million concurrent users during peak hours.
Availability	The system must be available 99.99% of the time (4.38 minutes downtime/month).
Performance	API responses should be under 300ms for 95% of requests.
Security	Passwords must be encrypted (e.g., bcrypt), and data should use HTTPS.
Usability	The app should be easy to navigate, responsive, and user-friendly.
Maintainability	The codebase should follow clean code principles for easy updates and debugging.
Logging and Monitoring	All service failures and anomalies should be logged and monitored via tools like Prometheus/Grafana.
Data Consistency	In case of distributed transactions (e.g., placing order + payment), ensure ACID or eventual consistency.
Localization	The app should support multiple languages (English, Hindi, Tamil, etc.).

HLD and LLD

Creating HLD (High-Level Design) and LLD (Low-Level Design) for your project is like building a roadmap from the big picture down to the nuts and bolts. Here’s a step-by-step guide to help you design both effectively

🧭 High-Level Design (HLD): Start with the Architecture

1. **Understand Requirements**
   - Functional: What should the system do? (e.g. user login, payment processing)
   - Non-functional: Scalability, performance, availability, etc.
2. **Identify Core Modules**
   Break down the system into services or components like authentication, order processing, analytics, etc.
3. **Define Interactions**
   How do components communicate (REST, gRPC, message queues)? Do they interact synchronously or asynchronously?
4. **Select Technologies**
   Choose databases, frameworks, messaging systems, security protocols, etc.
5. **Draw Architecture Diagrams**
   Include service boundaries, data flow, database access, external dependencies, caching, etc.
6. **Consider Scaling Strategies**
   Load balancers, replication, stateless services, and CDN if relevant.

🔧 Low-Level Design (LLD): Zoom into the Implementation

1. **Break Down Components**
   For each HLD module, define the internal classes, interfaces, and their responsibilities.
2. **Define Data Models**
   Use entity-relationship diagrams or detailed database schemas.
3. **Detail Business Logic**
   Write pseudocode or flowcharts that describe logic (e.g., order validation, transaction handling).
4. **API Specifications**
   Define request/response formats with headers, parameters, status codes.
5. **Handle Edge Cases & Failures**
   Define what happens in case of errors—timeouts, retries, fallback mechanisms.
6. **Diagrams**
   Create class diagrams, sequence diagrams, and interaction flows using tools like Lucidchart or PlantUML.

Realtime Example: Online Food Delivery Platform (like Swiggy/Zomato)

✅ High-Level Design (HLD)

🎯 Goal: Design a scalable food delivery platform that allows users to:

- Browse restaurants
- Place orders
- Track deliveries
- Make payments

🧱 Main Components:

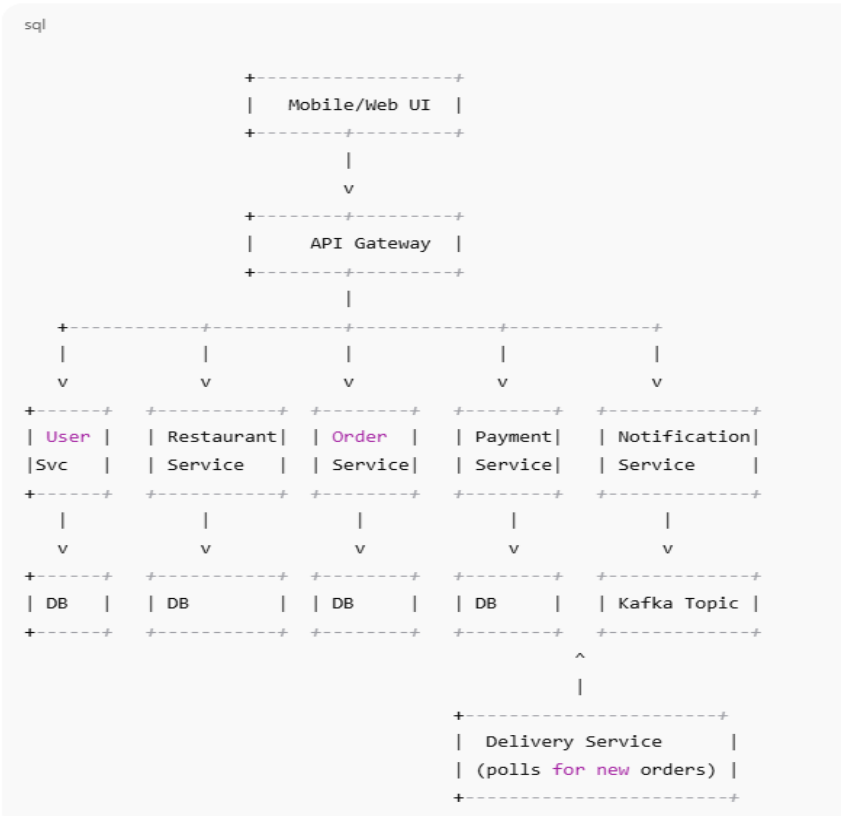
1. **User Service** – Registration, login, profile
2. **Restaurant Service** – List restaurants, menu management
3. **Order Service** – Place & manage orders
4. **Delivery Service** – Assign & track delivery agents
5. **Payment Service** – Handle transactions
6. **Notification Service** – Send updates via SMS/email/push
7. **API Gateway** – Entry point for all client requests
8. **Mobile/Web Clients** – Customer and delivery agent apps
9. **Database(s)** – Separate databases per domain (optional per service)

🌐 Technologies:

- Backend: Java/Spring Boot, Node.js
- Frontend: React Native / Flutter
- DB: PostgreSQL, MongoDB
- Messaging: Kafka (async events)
- Caching: Redis
- Storage: AWS S3 (e.g., for images)
- Monitoring: Prometheus + Grafana

🏗️ HLD Diagram:

sql



✅ Low-Level Design (LLD)

🎯 Zoom into: Order Service

📘 Use Case: Placing an Order

🛠️ API Contract:

```
css

POST /orders
Request:
{
  "userId": 101,
  "restaurantId": 99,
  "items": [
    { "itemId": 1, "quantity": 2 },
    { "itemId": 3, "quantity": 1 }
  ],
  "paymentMethod": "UPI"
}

Response:
{
  "orderId": 555,
  "status": "PENDING"
}
```

🔷 Class Diagram (Simplified)

```
sql

+-------------------+
| OrderController   |
+-------------------+
| +createOrder()    |
+-------------------+
         |
         v
+-------------------+       +-----------------+
| OrderService      |-----> | PaymentService  |
+-------------------+       +-----------------+
| +placeOrder()     |       | +initiate()     |
+-------------------+       +-----------------+
         |
         v
+-------------------+
| OrderRepository   |
+-------------------+
| +save(order)      |
+-------------------+

+-------------------+
| Order             |
+-------------------+
| -orderId          |
| -userId           |
| -restaurantId     |
| -items[]          |
| -status           |
+-------------------+
```

Order DB Schema (PostgreSQL)

orders table:

order_id	user_id	restaurant_id	status	created_at
555	101	99	PENDING	2025-06-29 10:00

order_items table:

id	order_id	item_id	quantity
1	555	1	2
2	555	3	1

Sequence Diagram: Place Order

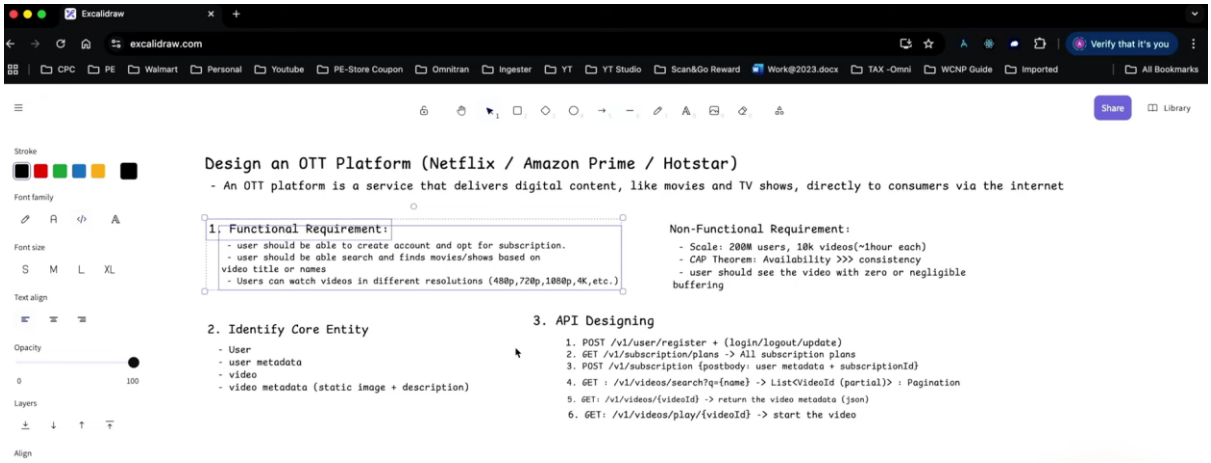
```rust
User -> API Gateway: POST /orders
API Gateway -> OrderService: createOrder()
OrderService -> RestaurantService: validateMenuItems()
OrderService -> PaymentService: initiate()
PaymentService -> Payment Gateway: charge()
Payment Gateway -> PaymentService: success
PaymentService -> OrderService: paymentSuccess
OrderService -> OrderRepository: save(order)
OrderService -> NotificationService: sendOrderConfirmation
OrderService -> User: Order placed successfully
```

Summary:

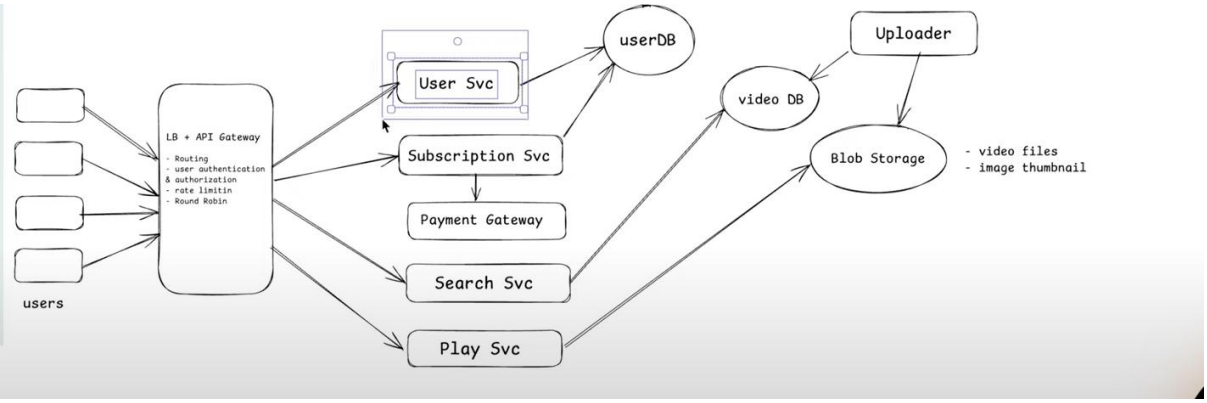
Element	HLD	LLD
Scope	Full system overview	Internal details of Order Service
Diagram	System architecture	Class diagram, Sequence diagram, DB schema
Deliverables	Components, communication flow	API contracts, method logic, data modeling
Tools (Optional)	Draw.io, Lucidchart, Whimsical	PlantUML, VS Code, ERD tools

System Design Netflix:

Functional and Non Functional Requirements



High level design:

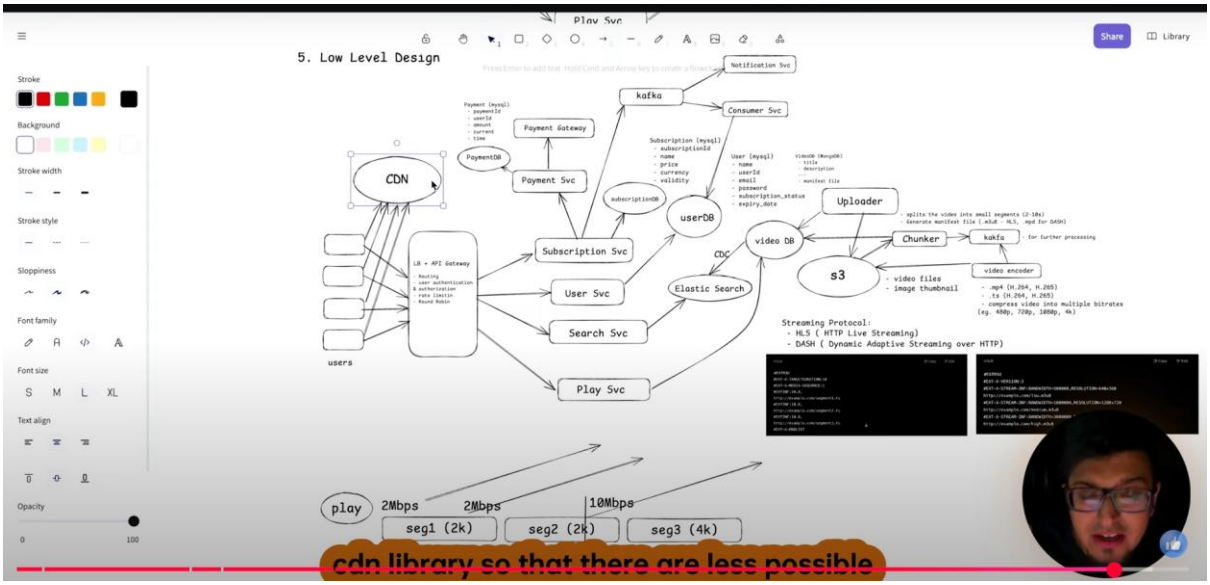


Uploader -> Will be handled by the Netflix system team

Video DB -> It will store the meta data about videos.

Blob Storage -> It is used to store the processed video.

LLD:

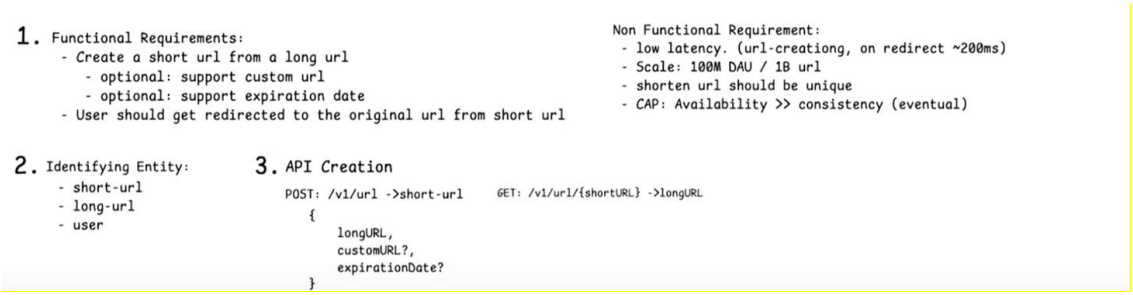


System Design Tiny URl:

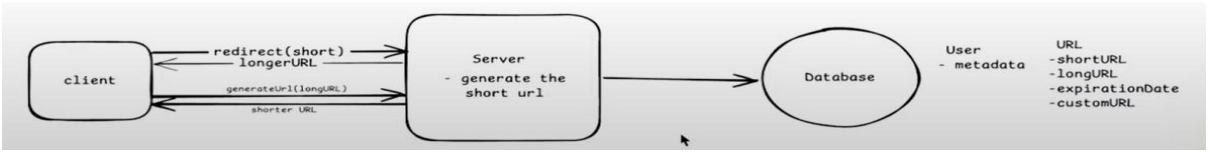
URL Shortener like Bit.ly

Bit.ly is a URL shortener service that lets you shorten and customize links.
eg: https://www.facebook.com/anindya.s.dasgupta/ => http://bit.ly/4kyLcpo

FR, NFR and API Design:



High Level Design:

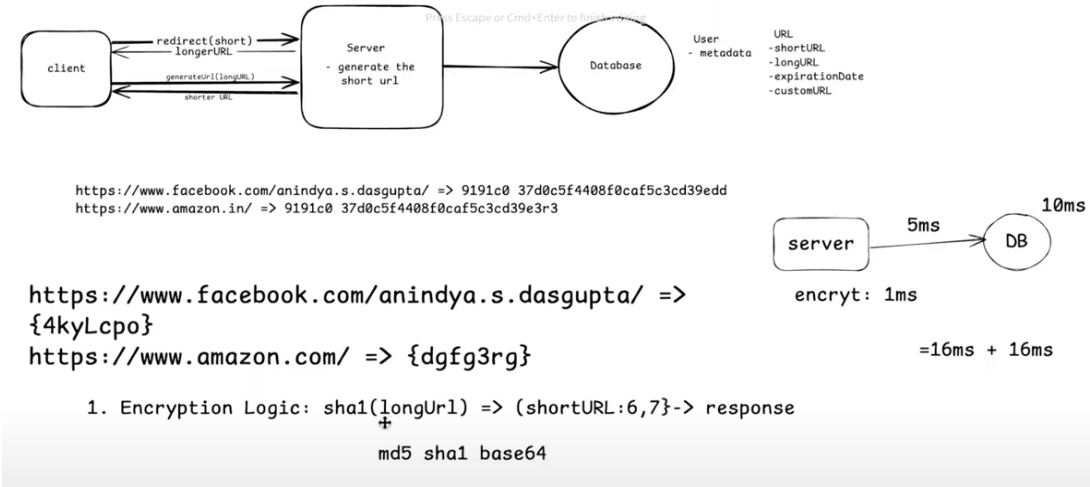


Here two data bases will be created,

User -> Holding the user information

URL – For holding the shorter and longer url.

Low Level Design:



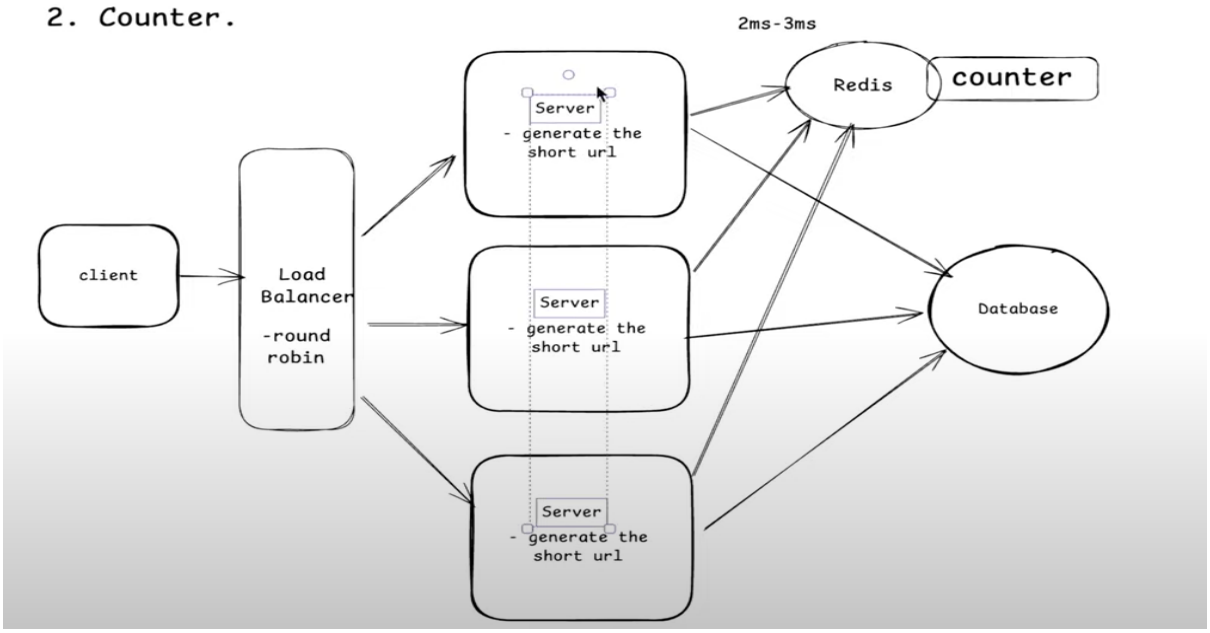
First Approach Encryption Logic:

As a first approach we took the encryption logic. As you see hash generated for url is longer and it will consume the space so instead of returning the entire hash will return first six chars. Here there a chance of getting the same six digits for another url, hence first we need to check the existing url in the data base if not exists returns the existing one. Hence it's a time consuming approach.

In the above example its taking 16ms to check the new url existence in database its taking 10 ms and for encryption its taking 1ms and to save the url into database its taking 5ms.

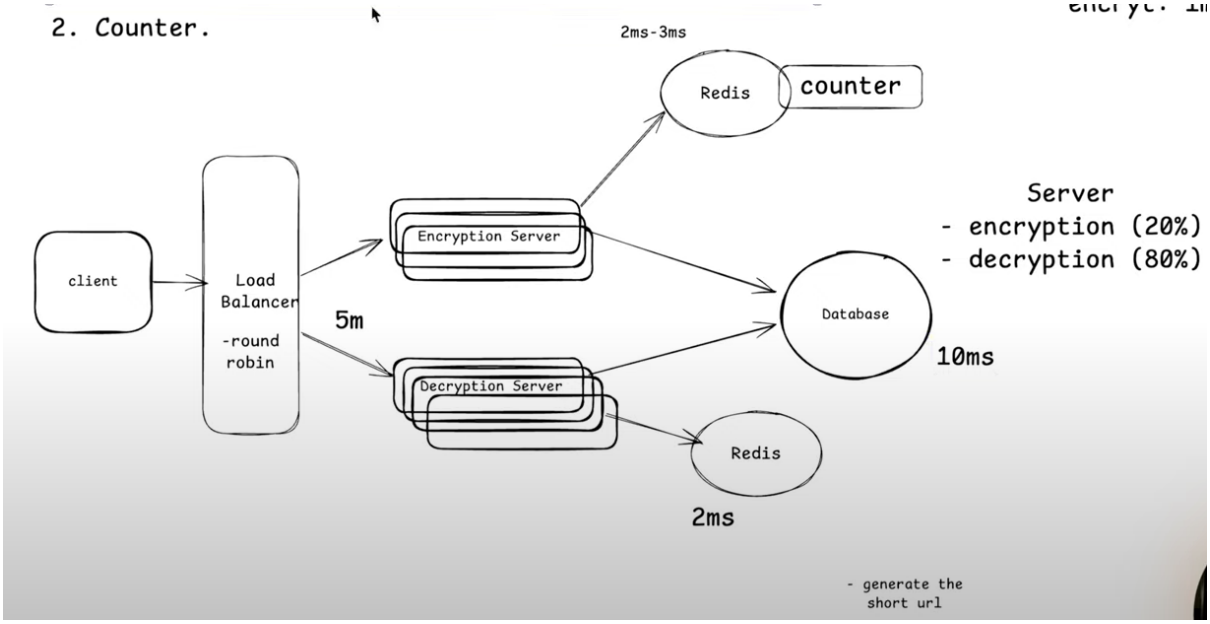
Second Approach is Counter:

In the counter approach we will use the microservices as we are receiving billions of requests per second. To distribute the traffic equally among the services we will follow the horizontal scaling approach with load balancer as front door approach. In this approach we will maintain the counter at Redis cache level to generate the unique url in all the servers. Here the counter value will be cached and based on the exsting counter value in session next url will be generated. Here we are maintain the counter at redis cache instead of the server local cache because we don't know load balancer send request to which server. If we maintain server level cache lets consider first two hits came to the server 1 then it will generate the url's with count 0 and 1. In the next hit if load balancer send the request to send server again counter started from 0, so there is a chance of duplicate url's. that's why we are maintain global cache with redis.



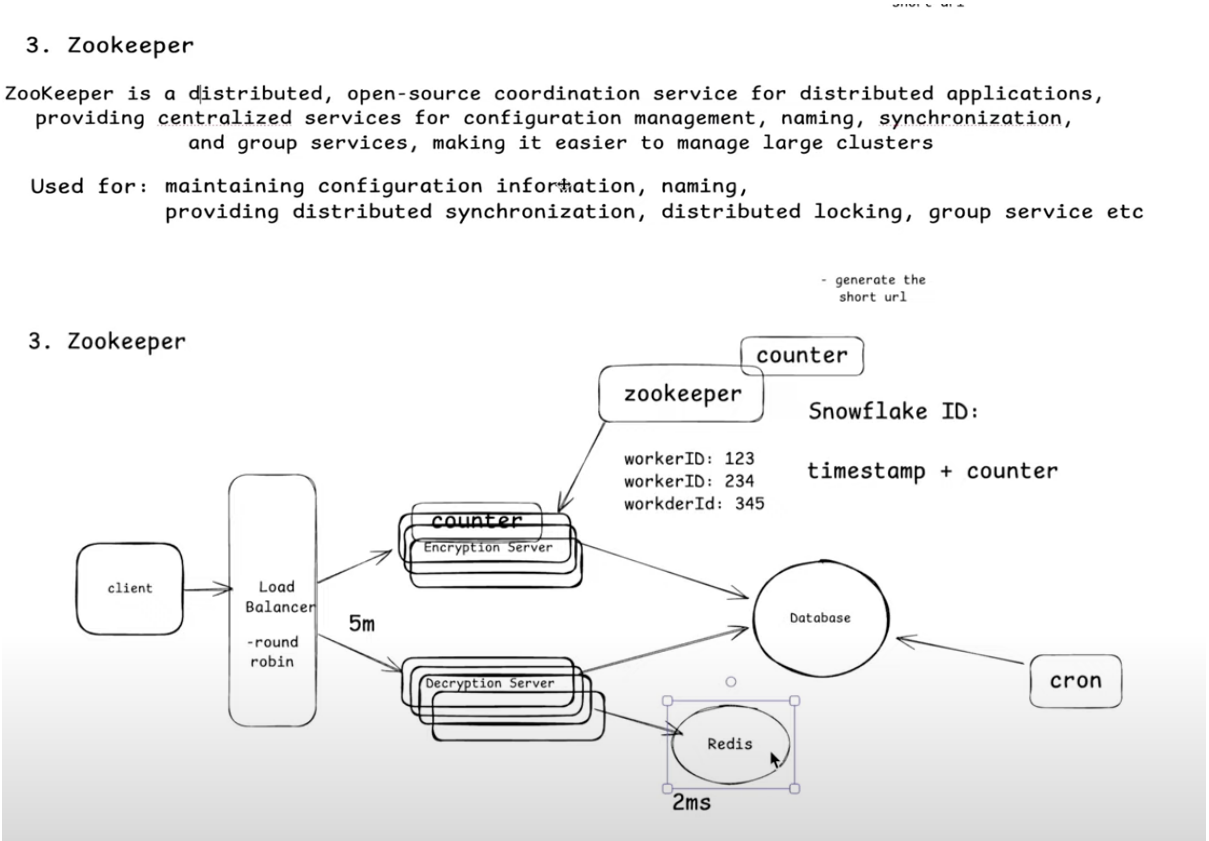
Here the problem is redis cache, because if redis node is single point of failure. If we maintain the redis cluster with multiple redis servers after serving few requests if any of the redis server is down again the counter starts from 0 and there is a chance of duplicate data.

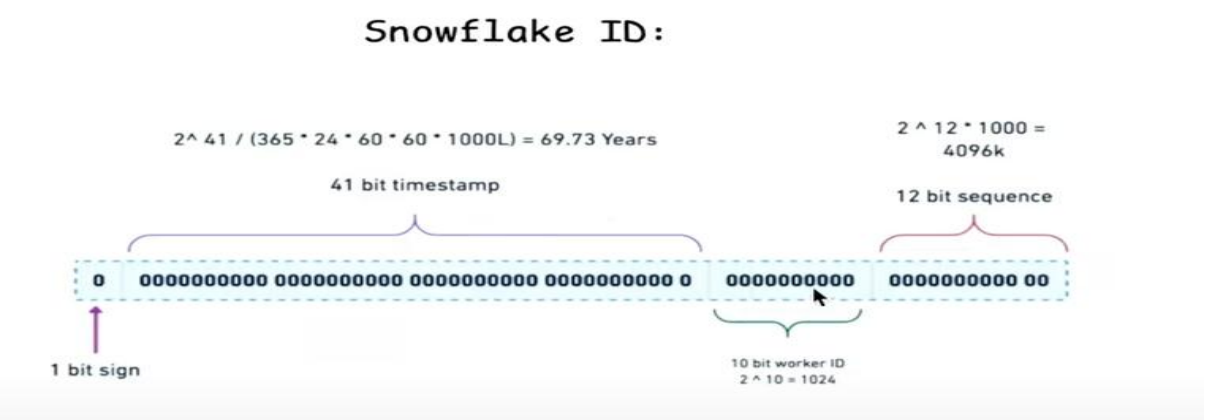
In the counter another approach instead of keeping the encryption and decryption at on place will add the separate micro services for Encryption and Decryption. Here as encryption as less traffic we can scale 3 instance encryption server and 6 instances of Decryption server. To reduce the latency at decryption time we will maintain redis cluster at decryption as well.



In this approach also problem is redis cache, because if redis node is single point of failure. If we maintain the redis cluster with multiple redis servers after serving few requests if any of the redis server is down again the counter starts from 0 and there is a chance of duplicate data.

Final Approach with Zookeeper:





In this approach we are replacing the redis cache cluster with Zookeeper. Here Zookeeper will manage the encryption and decryption cluster. In this case we will maintain the counter at Encryption server level. In this architecture zookeeper will generate the unique worker id(snow flake id) for each instance, with that unique id we can generate the unique tiny url for each request. Even in case anyone of the Encryption instance down the counter will be backed up at Zookeeper counter level. As we are not doing any interactions with zookeeper even if zookeeper fail we have backup counter at encryption server itself.

Clean Up:

As we are saving the billion of records in database and redis cache its not good to store the data for long time in database and redis cache. For this we should use cron job on daily basis to delete the expired records from database. In the redis cache at the configuration level we will use the TTL(Time To Live) for 30 days or 90 days to remove from cache.

For custom URL initially we check whether that url available at database if not will create new one.

For url redirection either will use 301 or 302.

301 means permeant redirection means while redirecting the in local cache it will be stored. So it will reduce the number of calls and redirection time.

302 means every time it will hit the decryption server and fetch the corresponding long or short url form cache or database then it will be redirected. If we want to track the number hits in the form of metrics 302 will be best.

As we are not storing much data it good to go with either MySQL or Postgres Database. But create index on the short url for better performance.

System Design for Messenger:

Design an Chat Server (Messenger / Whatsapp)

- User should be able to send and receive message from friends and groups in near real time

1. Functional Requirements:

1. User should be able to register/login
2. One-to-one messaging
3. Group messaging
4. Support both Text/Media message
5. Message history
6. Delivery/Read receipts

Non-Functional Requirements

1. Scale: 1 Billion users, 100msg/day ->100B msg/day (1kb*100B = 100TB/day)
2. CAP: Availability >> consistency
3. low-latency < 300ms
4. Highly reliable with no message loss

2. Core Entities

1. User
2. Group
3. Message/Chat

3. API Designing

1. POST /v1/user/register {PostBody: userName, email/phone...}
2. ...

One-To-One

1. WS: /v1/messages/send
2. GET /v1/chat/{userId}: List<Chat> :Pagination
3. GET /v1/message/{userId}/{receiverId}

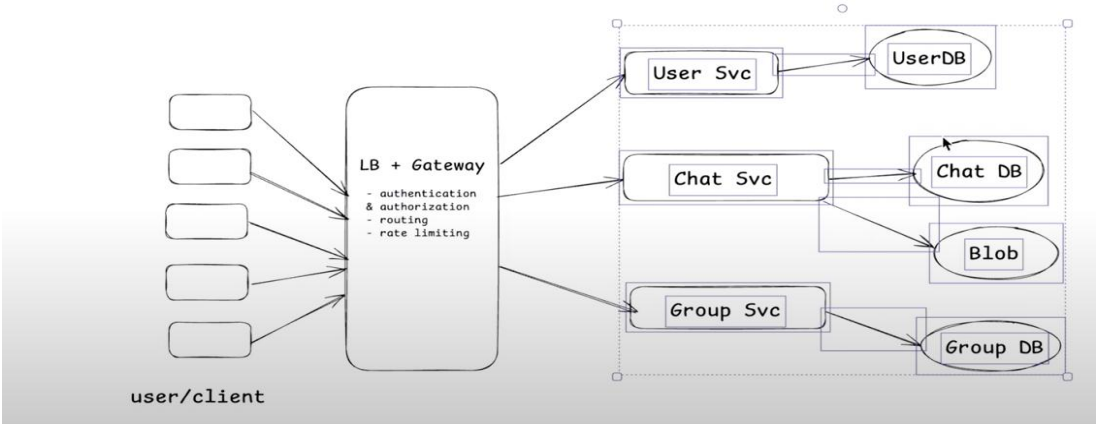
Group-Msg

1. POST /v1/groups/create
2. POST /v1/groups/{groupId}/add
3. POST /v1/groups/{groupId}/remove

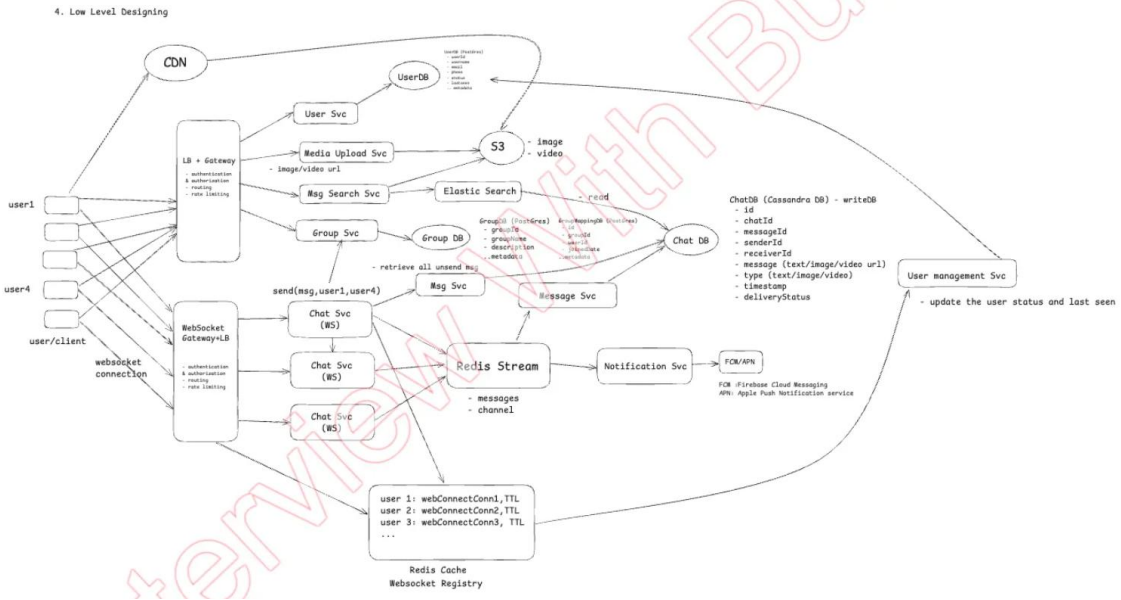
1. WS: /v1/messages/send
2. GET: /v1/groups/{groupsId}/message: Lazy Load

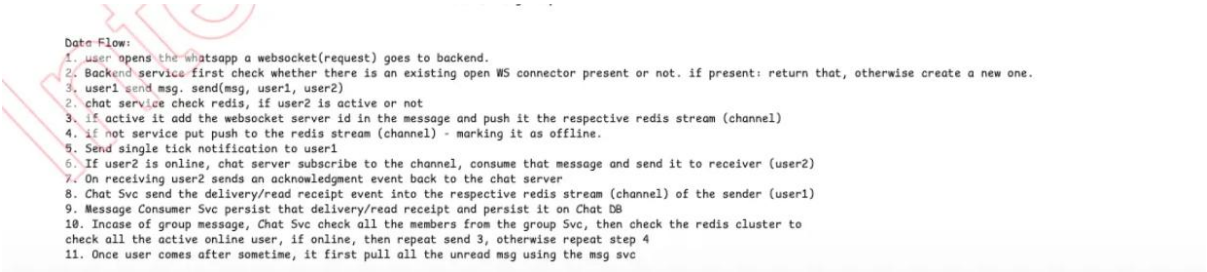
High Level Design:

3. High Level Designing



Low Level Design:





```
Data Flow:
1. user opens the whatsapp a websocket(request) goes to backend.
2. Backend service first check whether there is an existing open WS connector present or not. if present: return that, otherwise create a new one.
3. user1 send msg. send(msg, user1, user2)
2. chat service check redis, if user2 is active or not
3. if active it add the websocket server id in the message and push it the respective redis stream (channel)
4. if not service put push to the redis stream (channel) - marking it as offline.
5. Send single tick notification to user1
6. If user2 is online, chat server subscribe to the channel, consume that message and send it to receiver (user2)
7. On receiving user2 sends an acknowledgment event back to the chat server
8. Chat Svc send the delivery/read receipt event into the respective redis stream (channel) of the sender (user1)
9. Message Consumer Svc persist that delivery/read receipt and persist it on Chat DB
10. Incase of group message, Chat Svc check all the members from the group Svc, then check the redis cluster to
check all the active online user, if online, then repeat send 3, otherwise repeat step 4
11. Once user comes after sometime, it first pull all the unread msg using the msg svc
```

WebSocket is used for real-time communication between WhatsApp users. Each user's device establishes a persistent WebSocket connection to the WebSocket server, allowing for real-time message delivery.

Responsibilities:

- Maintain an **open connection** with each active (online) user.
- **Map users** to a port assigned to them on the WebSocket server.
- **Notify** users when they have a new message or event.

Uber/OLA System Design:

Functional Requirements:

User	Driver
Create Profile	Create Profile
Add start and end location	Show Availability
Show near by cabs/Bikes/Auto	Accept/Reject Trip
Show ETA and approximate price	Navigate to Pick Up Location
Book Cab	Start Ride
Make Payment	Navigate to Drop Up Location
Rate Driver	End Ride
See Past Trips	Payment
Raise complement	Rate Customer
Get current trip status	See Past Trips
Get particular past trip	

Non Functional Requirements:

- ✓ **Low Latency:** Latency refers to the time delay between a request being sent from one service and the response being received.

- ✓ **Highly Scalable:** It means the system can **handle increasing loads** (like users, data, or traffic) **efficiently** by **adding resources** (horizontally or vertically), **without performance degradation**.

✓ **Reliability:** Reliability in Microservices is all about ensuring that your distributed system continues to function correctly—even when individual components fail. Because microservices are loosely coupled and independently deployable, they introduce both flexibility and complexity. Here's how you can build reliability into them

Key Strategies for Reliability

- **Resilience Patterns**:
  - **Circuit Breaker**: Prevents a service from repeatedly calling a failing service, allowing it time to recover.
  - **Retry with Backoff**: Automatically retries failed requests with increasing delay to handle transient issues.
  - **Bulkhead**: Isolates failures by partitioning services so one failure doesn't cascade across the system.
  - **Timeouts**: Ensures that calls don't hang indefinitely, freeing up resources for other requests.

Summary

Reliability = Resilience + Fault Tolerance + Monitoring

A reliable microservices system ensures:

- Minimal downtime
- Controlled failure scope
- Smooth user experience, even under stress or failure

✓ **CAP Theorem:**

CAP stands for:

- **Consistency (C)**: Every read receives the most recent write.
- **Availability (A)**: Every request receives a (non-error) response.
- **Partition Tolerance (P)**: The system continues to operate despite network failures.

In distributed systems, you can only guarantee **two out of the three** at any given time.

Rider & Driver Matching

- **Priority**: **Availability over Consistency**
- Why? If a rider requests a ride, Uber must respond quickly—even if some backend services are temporarily out of sync.
- Trade-off: It may show a driver who just went offline or assign a ride that's already taken, but it avoids showing an error or making the user wait.

Location Updates

- **Priority**: **Availability**
- GPS updates are frequent and can tolerate slight inconsistencies. It's better to show a slightly outdated location than no location at all.

Payments & Trip Completion

- **Priority**: **Consistency**
- Financial transactions must be accurate. Uber ensures that trip completion and billing are consistent, even if it means retrying or delaying the operation slightly.

1. **Explain system design formula for calculating the requests received for year, Month, Hour and seconds?**

   To estimate the number of requests your system might need to handle over different timeframes (like per year, month, hour, or second), you can use a scalable forecasting formula based on peak and average traffic patterns. Here's a practical breakdown

Formula:

```
Daily Requests = Number of Active Users × Requests per User per Day
```

Let's define some terms:

- U = Number of users
- R_d = Average requests per user per day
- From this, you can derive the requests for other time periods.

Breakdown by Time Unit:

Yearly Requests:

$$R_{year} = U \times R_d \times 365$$

Monthly Requests (approx):

$$R_{month} = U \times R_d \times 30$$

Daily Requests:

$$R_{day} = U \times R_d$$

Hourly Requests:

$$R_{hour} = \frac{R_{day}}{24} = \frac{U \times R_d}{24}$$

Per Second Requests (RPS):

$$R_{second} = \frac{R_{hour}}{3600} = \frac{U \times R_d}{24 \times 3600}$$

Example:

Active users: 100,000
Each makes 10 requests/day

Requests per year: 100,000 * 10 * 365 = 365,000,000
Requests per Month: 100,000 * 10 * 30 = 30,000,000
Requests per week: 100,000 * 10 * 7 = 70,00,000
Request per day: 100,000 * 10 = 10,00,000
Requests per hour: 10,00,000/24 = 41,667 (Requests per day/24 Hours)
Requests per second: 41,667/3600 = 12 (Request per hour/3600 Seconds)

Here 24 Hours means 1 day in requests per hour calculation
Here 3600 Seconds means 1 hour is equal to 60 minutes that is 60x60 seconds = **3600 seconds**

Another Example:

🧠 Example Calculation:

Assume:

- $U = 10$ million users
- $R_d = 5$ requests per user per day

Now compute:

- **Yearly**:

$$10^7 \times 5 \times 365 = 18.25 \text{ billion requests/year}$$

- **Monthly**:

$$10^7 \times 5 \times 30 = 1.5 \text{ billion requests/month}$$

- **Daily**:

$$10^7 \times 5 = 50 \text{ million requests/day}$$

- **Hourly**:

$$50 \text{ million} \div 24 \approx 2.08 \text{ million/hour}$$

- **RPS**:

$$2.08 \text{ million} \div 3600 \approx 578 \text{ requests/sec}$$

Now calculating the Servers required based on the requests receiving:

✅ Step 3: Core Formula

$$\text{Number of Servers Required} = \left\lceil \frac{RPS_{peak}}{C_s \times \text{Utilization Factor}} \right\rceil$$

Where:

- **Utilization Factor** = To avoid 100% usage (usually 0.7–0.8 to leave buffer)
- $\lceil \cdot \rceil$ means round up (always overestimate to ensure reliability)

- C_s = How many requests **a single server can handle per second** (i.e., server's RPS capacity)

🧠 Example

Assume:

- Peak traffic: $RPS_{peak} = 5000$
- Server capacity: $C_s = 100$ RPS
- Utilization: 80% (i.e., 0.8)

$$\text{Servers} = \left\lceil \frac{5000}{100 \times 0.8} \right\rceil = \left\lceil \frac{5000}{80} \right\rceil = \lceil 62.5 \rceil = 63 \text{ servers}$$

2. Can you explain how autoscaling can be implemented?

Auto-scaling is a key design pattern in **modern cloud architecture** that helps you automatically **add or remove servers** (compute resources) based on real-time load (e.g., traffic, CPU, memory).

🚀 What Is Auto-Scaling?

Auto-scaling ensures that your system:

- **Scales up** (adds more instances) when demand increases
- **Scales down** (removes instances) when demand decreases

It helps with:

- **High availability**
- **Cost optimization**
- **Performance under load**

🧰 Types of Auto-Scaling

Type	Description
Horizontal Scaling	Add/remove instances (e.g., 3 → 6 EC2 servers)
Vertical Scaling	Increase/decrease instance power (e.g., 2 vCPU → 4 vCPU)
Scheduled Scaling	Scale based on known traffic patterns
Dynamic Scaling	Scale based on real-time metrics (CPU, RPS, etc.)

⚙️ How Auto-Scaling Works (Step-by-Step)

1. Choose Metrics to Monitor

Auto-scaling decisions are based on these metrics:

- **CPU Utilization** (most common)
- **Memory Usage**
- **Network I/O**
- **Request Count or RPS**
- **Custom Metrics** (e.g., Queue size, DB load)

---

2. Define Scaling Policies

Set thresholds for when to scale up/down.

Example:

- Scale **up** if average CPU > 70% for 2 minutes
- Scale **down** if CPU < 30% for 5 minutes

---

3. Set Min/Max Limits

Prevent infinite scaling:

- **Min Instances**: Always keep 2 servers running
- **Max Instances**: Don't exceed 50 to control cost

4. Integration with Load Balancer

- Use a **Load Balancer** (e.g., AWS ALB, GCP Load Balancer, Nginx) to route traffic to new instances
- Auto-scaling group should **register new instances** to the load balancer

---

5. Graceful Health Checks

- Use **health checks** to verify that new instances are ready before routing traffic to them

🧩 Integration with Microservices

- Each **microservice** can scale independently
- Deploy via **Docker/Kubernetes**
- Use **Kubernetes HPA**:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-service
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

3. Can you explain how autoscaling can be implemented in multi regional cluster?

Implementing **auto-scaling in a multi-regional cluster** adds complexity but greatly improves **resilience, latency**, and **global availability**.

🌎 What Is a Multi-Regional Cluster?

A **multi-regional (or multi-zone)** cluster spans across multiple geographical regions (e.g., US, Europe, Asia). It's often used in:

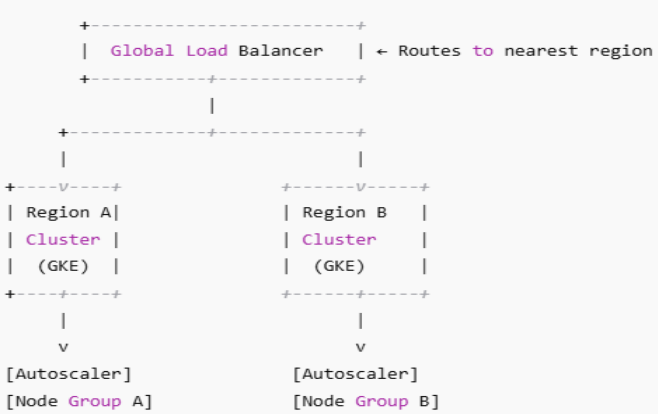
- **Kubernetes (GKE, EKS, AKS)**
- **Cloud VMs (AWS EC2, GCP Compute Engine)**
- **Serverless platforms**

🏗️ Components Involved

Component	Role
Load Balancer (Global)	Routes traffic to nearest healthy region or zone
Regional Clusters	Each region has its own cluster or node group
Auto-scaler (per region)	Handles scaling independently in each region
Service Mesh (Optional)	Manages inter-region service communication (e.g., Istio)

✅ High-Level Design

pgsql



⚙️ Step-by-Step Implementation

✅ 1. Deploy Clusters in Multiple Regions

For example, in Kubernetes:

- Deploy one cluster in `us-central1`
- Another in `europe-west1`

Or use **GKE Regional Clusters** that replicate across multiple zones in a region.

✅ 2. Enable Auto-Scaling in Each Region

In **Kubernetes**, configure:

a. **Horizontal Pod Autoscaler (HPA)**:

Scales pods based on CPU, memory, or custom metrics.

yaml

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: web-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web-app
  minReplicas: 3
  maxReplicas: 15
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

b. **Cluster Autoscaler**:

- Auto-scales nodes (VMs) in the region
- Enabled per-node pool or node group
- Supported in **GKE, EKS, AKS**

✅ 3. Global Load Balancing

Use a **global load balancer** to:

- Detect region health
- Route user to the **nearest healthy region**
- Perform failover if a region is down

Examples:

- **GCP**: Global HTTP(S) Load Balancer
- **AWS**: Route 53 + Application Load Balancer
- **Cloudflare** / **Akamai**: Global traffic managers

✅ 4. Sync Traffic and Deployments

Ensure:

- **Same app version** is deployed across all regions
- Config, secrets, and images are consistent (use CI/CD pipelines)

Use tools like:

- ArgoCD / Flux for GitOps
- Helm charts for multi-region support

✅ 5. Observability

- Use **centralized monitoring/logging** (Prometheus + Grafana, Datadog, CloudWatch)
- Collect per-region metrics and configure alerts:
  - CPU, memory
  - Request count / latency
  - Pod health
  - Region outage

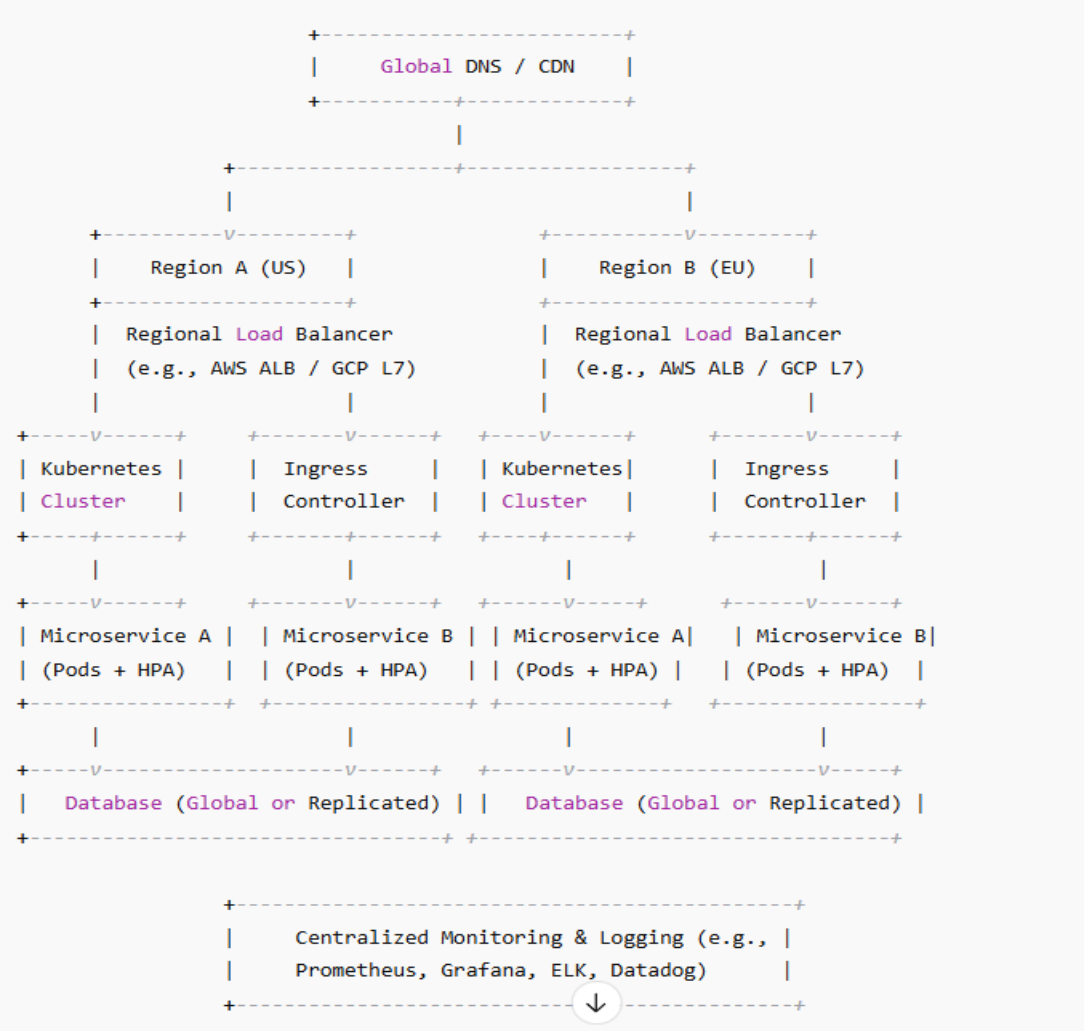
🔄 Traffic Flow Example

1. User in **Germany** hits global DNS
2. DNS resolves to **europe-west1 load balancer**
3. Load balancer routes to healthy pod in region
4. If region goes down, failover to **us-central1**

✅ Summary

Component	Description
Global LB	Routes traffic to nearest healthy region
Regional Cluster	Manages scaling locally via HPA + node autoscaler
HPA	Scales pods based on real-time metrics
Cluster Autoscaler	Adds/removes nodes based on pod demand
CI/CD	Keeps deployments in sync across regions
Observability	Monitors per-region performance and health

4. Explain micro services multi regional Kubernetes architecture?



🧩 Component Explanation

1. 🌐 Global DNS / CDN

- Routes users to the **nearest healthy region**
- CDN (e.g., Cloudflare, Akamai) caches static assets close to the user
- Supports **geo-routing** and **failover**

---

2. 🌎 Regions (e.g., US & Europe)

Each region is independent, but **synchronized**. It contains:

✅ **Regional Load Balancer**

- Receives traffic from the global DNS
- Routes to appropriate services within the cluster
- Examples:
  - AWS ALB / NLB
  - GCP HTTPS Load Balancer

✅ **Kubernetes Cluster**

- Hosts all microservices
- Cluster spans across multiple **zones** for HA

✅ **Ingress Controller**

- NGINX, HAProxy, or cloud-native ingress (like GKE Ingress)
- Handles routing to services internally

✅ **Microservices (Pods)**

- Stateless services managed by **Deployments**
- Autoscaled using:
  - **Horizontal Pod Autoscaler (HPA)** → based on CPU, memory, custom metrics

---

✅ **Cluster Autoscaler**

- Automatically adds/removes nodes based on pending pod demand

3. 🧱 Database Layer

Options:

- **Multi-region database** (like Google Spanner, DynamoDB global tables)
- **Replication setup**:
  - **Primary in one region**, read replicas in others
  - Use **eventual consistency** models for high throughput

---

4. 📊 Monitoring & Logging

Centralized observability with tools like:

- **Prometheus** + **Grafana** for metrics
- **ELK Stack (Elasticsearch** + **Logstash** + **Kibana)** or **Datadog**
- Logs and metrics aggregated across regions

---

5. 🔁 CI/CD & GitOps

- Use tools like **ArgoCD** or **Flux** for deploying apps consistently across clusters
- Pipeline stages per region

🧠 Key Considerations

Topic	Details
High Availability	Use multiple zones per region; replicate across regions
Latency	Users connect to geographically closest cluster
Failover	DNS-based or load balancer failover to healthy region
Cost	Multi-region setup is more expensive; use autoscaling
Data Consistency	Choose consistency model carefully (strong vs eventual)
Secrets Management	Sync secrets using tools like Vault or Sealed Secrets

---

🛡️ Optional Enhancements

- **Service Mesh**: (Istio / Linkerd) for secure, observable service-to-service communication
- **API Gateway**: For versioning, rate-limiting, auth
- **Queue/Broker**: Kafka (multi-region), SQS, Pub/Sub for async communication
- **Feature Flags**: Canary and gradual rollouts across regions

5. Explain the better databases in micro services system design?

🔍 Comparison Table (Read + Write Focus)

DB	Type	Read Speed	Write Speed	Consistency	Scaling	Best For
PostgreSQL	Relational	High	High	Strong (ACID)	Vertical / Citus	Most OLTP apps
Aurora	Managed SQL	Very High	Very High	Strong	Auto	Cloud-native SQL apps
CockroachDB	Distributed SQL	High	High	Strong	Horizontal	Global apps with SQL needs
MongoDB	Document	High	High	Tunable	Horizontal	Flexible schema, quick prototyping
DynamoDB	Key-Value	High	Very High	Tunable	Auto	Scalable, serverless apps
ScyllaDB	Wide-column	High	Very High	Tunable	Horizontal	High-volume data ingestion
RedisJSON	In-memory	Very High	Very High	Tunable	Limited (RAM)	Session/cache/temp data

Apache **Cassandra** is a highly scalable, distributed NoSQL database designed for **write-heavy, high-availability, and fault-tolerant systems**. It's ideal for applications that require **fast, consistent writes**, **high uptime**, and **massive data volumes** across multiple regions or data centers.

✅ Core Uses of Cassandra Database

1. 🚀 Write-Heavy Applications

- Cassandra excels at **high-throughput writes** with minimal latency.
- Great for real-time ingest of logs, telemetry, metrics, and events.

Examples:

- Logging systems
- Telemetry collection from devices
- Application usage tracking

6. How to scale the system from 100 users to 1 million users support?

Scaling a system from **100 to 10 million users** requires **gradual architectural evolution**, **infrastructure upgrades**, and **strong engineering practices**. Here's a structured roadmap to scale efficiently:

🚦 1. Foundational Setup (0 - 1K users)

Key Focus: Get product-market fit, build fast, deploy fast.

- **Monolithic architecture** (Spring Boot, Rails, Django).
- Host on a single instance (e.g., EC2, Heroku).
- Single database (PostgreSQL, MySQL).
- CI/CD setup, basic monitoring (New Relic, Datadog).

Scaling a system from **100 to 10 million users** requires **gradual architectural evolution**, **infrastructure upgrades**, and **strong engineering practices**. Here's a structured roadmap to scale efficiently:

🚦 1. Foundational Setup (0 - 1K users)

Key Focus: Get product-market fit, build fast, deploy fast.

- **Monolithic architecture** (Spring Boot, Rails, Django).
- Host on a single instance (e.g., EC2, Heroku).
- Single database (PostgreSQL, MySQL).
- CI/CD setup, basic monitoring (New Relic, Datadog).

⚙️ 3. Scaling Core (100K - 1M users)

Key Focus: Scalability, availability, performance.

- Adopt **microservices** or **modular monolith**.
- API Gateway (e.g., Kong, AWS API Gateway).
- **Database sharding** or horizontal partitioning.
- Separate read/write DB paths.
- Asynchronous communication: Kafka, RabbitMQ.
- Circuit breakers, retries (e.g., Resilience4j).
- Strong observability (Grafana, Prometheus, ELK stack).
- **Feature flags**, **A/B testing**, blue/green deployments.

🏗️ 4. Enterprise-Scale (1M - 10M users)

Key Focus: Global availability, fault-tolerance, efficiency.

- **Multi-region deployment** (AWS multi-AZ/multi-region).
- Globally distributed CDN + edge caching.
- Distributed databases (e.g., CockroachDB, DynamoDB, Spanner).
- **Polyglot persistence** (e.g., MongoDB for documents, Redis for sessions).
- Global load balancing (Route53, GSLB).
- Use **Kubernetes** or orchestration platforms.
- Zero-downtime deployments, canary releases.
- Data lake or stream processing (e.g., Kafka + Flink + S3).
- Security hardening: IAM, rate-limiting, DDoS protection (WAF).